Albert User's Guide

Written on February 7, 1996 for Albert version 3.0 by
David Pokrass Jacobs, Sekhar V. Muddana, A. Jefferson Offutt, Kurti Prabhu, David Lee, Trent Whiteley
(Department of Computer Science, Clemson University)

In 2019 (last minor revision February 12, 2023) adapted for Albert version 4.0M7, and merged with former online help by
Pasha Zusmanovich
pasha.zusmanovich@gmail.com

1. Introduction

Albert is an interactive research tool to assist the specialist in the study of nonassociative algebras. This document serves as a technical guide to Albert. We refer the reader to [JMO] for a more casual tutorial[†]. The main problem addressed by Albert is the recognition of polynomial identities. Roughly, Albert works in the following way. Suppose a user wishes to study alternative algebras. These are algebras defined by the two polynomial identities $(yx)x - y(xx)$ and $(xx)y - x(xy)$, known respectively as the right and left alternative laws. In particular, the user wishes to know if, in the presence of the right and left alternative laws, $(a,b,c)\circ[a,b]$ is also an identity. Here (a,b,c) denotes $(ab)c - a(bc)$, $[a,b]$ denotes $ab - ba$, and $x \circ y$ denotes $xy + yx$. The user first supplies Albert with the right and left alternative laws, using the **identity** command. Next, the user supplies the *problem type*. This refers to the number and degree of letters in the target polynomial. For example, in this problem, each term of the target polynomial has two a's, two b', and one c, and so the problem type is $2a2b1c$. This is entered using the **generators** command. It may be that over certain fields of scalars the polynomial is an identity, but over others it is not an identity. Albert allows the user to supply the field of scalars, but currently the user must select either a Galois field $\mathsf{GF}(p)$ in which p is a prime less than $2^{63}-1$, or the field of rational numbers $\mathbb{Q}$[‡]. This is done using the **field** command. If no field is entered, the default field GF(251) is chosen.

In deciding whether a given polynomial is an identity or not, Albert internally constructs a certain homomorphic image of the free algebra. It is not necessary that the user understand the theory of free algebras, nor is it necessary to understand the algorithms Albert employs to create them (which are described in [HJ1]). The user need only be aware that Albert builds a multiplication table for this free algebra. The user instructs Albert to begin the construction using the **build** command. Once this construction has been completed, the user can query whether the polynomial $(a,b,c)\circ[a,b]$ is an identity using the **polynomial** command. In fact, the user can ask Albert about any homogeneous polynomial $p(a,b,c)$ having most two a's, two b's and one c in each term. For example, the polynomial $(a,b,(a,b,c))+[b,a](a,b,c)$ could also be tested. With Albert, polynomials and identities may be entered from the keyboard using associators, commutators, and much of the familiar notation used in nonassociative ring theory. Since the standard keyboard does not have the symbol $\circ$, we use $*$ to denote the Jordan product. See the section entitled *Polynomial Expression Language* for a complete description of how polynomials and identities can be entered in Albert.

[†]Valid for version 3.0, largely valid for the current version.

[‡]With a little programming effort, Albert can use any field as long as it has a computer implementation with a C/C++ interface.

Albert has a small set of commands, and meaningful research can be conducted using only the **identity**, **generators**, **build**, and **polynomial** commands. These commands, and others, are described in detail in this guide. The user who wishes to quickly learn the system is advised to first try these commands. Thus, the typical user supplies Albert with the following input:

- A set I of polynomials whose members are assumed to be identities.
- A problem type as described above.
- A field F of scalars.

In this document "polynomial" always means a nonassociative polynomial. These objects together implicitly define a fourth object, namely the free nonassociative algebra. In this document we refer to these four objects collectively as a *configuration*. Various commands may alter or delete certain objects of a configuration. Typically, Albert spends much of its time constructing a multiplication table. But once a table has been constructed, Albert can quickly decide if a polynomial or group of polynomials are identities.

All polynomials and defining identities must be homogeneous. That is, each must be expressible as a linear combination of words each having the same degree in each variable. This restriction is not very severe, since any nonhomogeneous polynomial can be replaced by its homogeneous parts. This replacement will not affect the results given a sufficiently high characteristic for the field. Albert internally linearizes any defining identity that is not multilinear. Thus the right alternative identity is always interpreted as $(yx)z - y(xz) + (yz)x - y(zx)$.

However, if a defining identity is not multilinear, the user is advised *not* to linearize it before entering it, but rather enter it in its non-multilinear form. In most cases, this allows Albert to treat it more efficiently, since the identity's symmetry can be exploited. In general, the larger the degree of the problem type, the more memory Albert requires. Given two problems involving the same degree and the same defining identities, Albert will cope best with the one having fewer letters. The defining identities influence the problem, too. Defining identities such as the commutative law (as in the case of Jordan algebras) and the anticommutative law "drive down" the dimension of the free algebra, thus enabling larger problems to be solved than might otherwise be possible. If Albert is unable to complete a problem having degree n, adding additional identities of degree less than n may allow Albert to finish.

The program owes its name to A.A. Albert whose work was pioneering in nonassociative ring theory. It was designed and implemented at Clemson University by David P. Jacobs, Sekhar Muddana, and Jeff Offutt. Kurti Prabhu also helped during the early design. Subsequent features have been implemented by David Lee and Trent Whiteley. The idea of constructing free nonassociative algebras was motivated by several papers of E. Kleinfeld. Irvin Hentzel has also assisted in the program's development through helpful discussions.

Finally, a word of caution. Like any program, the possibility is high for errors. Please report any suspected bugs or general comments.

2. Command Language

The commands available with Albert are: **identity**, **remove**, **generators**, **field**, **build**, **display**, **polynomial**, **xpand**, **type**, **view**, **save**, **quit**.

Each command begins with a keyword, and the user can use any initial substring of the keyword. For example, the **identity** command can be used by typing **i**, **id**, etc. Every command is terminated by a carriage return. Some commands (e.g. **identity**, **polynomial**) require a polynomial as an argument. This polynomial may at times exceed the screen-width. In such cases, the user can continue on the next line by terminating the preceding line with a backslash (\).

2.1. **identity command.**

`identity` *polynomial*

Examples:

```
identity (x,x,[y,x])
i (x,(x,(x,y,x),x),x)
```

This command appends the polynomial to the current set of identities. Albert assigns a unique number to the entered identity for future use. Entering a new identity destroys any existing multiplication table in memory.

- Arguments: *polynomial* as described in the section *Polynomial Expression Language*. Names defined in the configuration file (see the section *Configuration file*) can also be used to enter a polynomial. The entered polynomial must be homogeneous.
- Errors: malformed or non-homogeneous polynomial.

2.2. **remove command.**

`remove` *number*|∗

This command removes one or more identities from the current set of identities. For example,

```
remove 2
```

would remove the identity whose number is 2. The remaining identities are renumbered after deletion of the identity. An asterisk (∗) can be used in place of *number* to remove all existing identities:

```
r *
```

The **remove** command will destroy a resident multiplication table, if present.

- Arguments: the number of the identity, or ∗
- Errors: invalid number.

2.3. **generators command.**

`generators` *problem_type*

Before beginning the construction of an algebra, Albert must know what the generators will be, and the degrees of the generators. This information is referred to as the problem type, and the generators command is used to define it. This problem type is stored in the current configuration. Entering a new problem type destroys any existing problem type and any multiplication table, if present. For example,

```
generators aabcc
gen aabcc
g 2ab2c
```

- Arguments: *problem_type* is a string of lower-case letters indicating the generators and degrees used in the problem. For example, `aabcc` indicates that the algebra to be built will be generated by a, b, and c, and will be spanned by words in these letters having at most two a's, one b, and two c's. This word must have degree at least two. This can also be entered in abbreviated form as `2a1b2c`, `2ab2c`, `b2a2c`, etc.

2.4. **field command.**

`field` *number*|`Q`

This command changes the field of scalars, and this information is stored as part of the current configuration. When the field is changed, any resident multiplication table is destroyed. For example,

```
field 17
```

will cause subsequent algebras to be constructed over the field GF(17). When Albert is first entered, the default field GF(251) is selected. This field remains in effect until the field is changed.

- Arguments: *number* must be a prime p less than $2^{63}-1$, in which case the field is changed to GF(p); if Q is specified, the field is changed to $\mathbb{Q}$.
- Errors: *number* out of range or not prime.

2.5. **build command.**

`build`

This command invokes Albert to begin construction of the algebra defined by the current configuration. Albert constructs the algebra using the current set of identities, problem type, and field stored in the current configuration. Status information is printed during the construction. An old multiplication table is destroyed.

- Arguments: none.
- Errors: problem type not defined, or memory overflow during the construction.

2.6. **display command.**

`display`

Typing `display` causes Albert to display the current set of defining identities, field, problem type, and information about the multiplication table, if present.

2.7. **polynomial command.**

`polynomial` *polynomial*

After a multiplication table has been constructed using **build**, this command may be used to test whether the given polynomial is zero in the resident algebra. For example, typing

```
p 2((ba)a)a + ((aa)a)b - 3((aa)b)a
```

might cause Albert to respond with:

```
Polynomial is not an identity.
```

- Arguments: *polynomial* has to be homogeneous.
- Errors: invalid or non-homogeneous polynomial, nonexistent table, polynomial incompatible with current problem type.

2.8. **xpand command.**

`xpand` *polynomial*

This command is used to see the expanded form of a nonassociative polynomial. For example, typing

```
xpand (x,y,z)
```

would cause Albert to respond with:

```
(xy)z - x(yz)
```

The command is used for information purposes only and does not affect the current configuration.

- Arguments: *polynomial* has to be homogeneous.
- Errors: invalid or non-homogeneous polynomial.

2.9. **type command.**

`type` *word*

Every nonassociative word has a particular association type. These association types are numbered by Albert. For example, the association type of the word $((ab)c)d$ is 1, while the association type of the word $(ab)(cd)$ is 3. Thus, typing

```
t (ab)(cd)
```

would cause Albert to respond with:

```
The association type of the word = 3.
```

This command prints the number of the association of the argument. Usually this is not important, unless the user wishes to use the W (artificial word) operator (see the section *Polynomial Expression Language*).

- Arguments: nonassociative *word* like $a((ac)b)$. The letters and their order are not important.
- Errors: invalid *word*.

2.10. **view command.**

`view b|m [gap]`

This command prints the basis table or the multiplication table to the screen, provided they exist. The first argument specifies the table to be output (`b` – basis table, `m` – multiplication table). For example, typing

`view b`

will output the current basis table to the screen.

If the first argument is `m`, and the second, optional, argument is specified, the multiplication table will be printed in the GAP format rather than in a human-readable format. For example, loading the output of the command

`view m gap`

to GAP will recreate the algebra constructed by Albert as a GAP object.

- Arguments: first – `b` or `m`, second (optional) – `gap`.

2.11. **save command.**

`save b|m [gap]`

This command saves the basis table or the multiplication table to a file, provided the table already exist. After typing the **save** command, the user will receive the following prompt:

```
File Name -->
```

The user may enter a directory path along with the file name. If a file name is entered, it will be written to the current directory. The arguments have the same meaning as for the **view** command. For example, typing

```
save m
File Name --> Mult.table
```

will cause the multiplication table to be written to `Mult.table` in the current directory.

- Arguments: first – `b` or `m`, second (optional) – `gap`.

2.12. **quit command.**

`quit`

This command exits the user from Albert. Any multiplication table or other information is lost.

2.13. **Summary of commands.**

`identity` *polynomial*	enter a defining identity
`remove` *number*`|*`	remove a defining identity
`generators` *word*	specify the problem type
`field` *number*`|Q`	change the current field of scalars
`build`	build a multiplication table
`display`	display the current configuration
`polynomial` *polynomial*	query if the polynomial is an identity
`xpand` *polynomial*	expand the polynomial
`type` *word*	determine the association type
`view b|m [gap]`	view basis or multiplication table
`save b|m [gap]`	save basis or multiplication table
`quit`	quit from Albert

3. Basis table and multiplication table

There are two important structures created by the **build** command. These are the basis table and the multiplication table. These tables can be seen using **view**, or stored using **save**. The basis table contains a list of elements that form a basis in the free algebra that was constructed. Under Albert's method, basis elements will always be words in the original generators. Shown below is the basis table for the 26-dimensional right alternative algebra using 2a's and 2b's. There are five columns. The first of these is simply the number by which the basis element is referred. The second and third columns indicate how the basis element factors into a product of two lower degree basis elements. The fourth columns indicates the type (degrees in each generator) of the element. The fifth column shows the basis element as a nonassociative word. For example, the table indicates that basis element #25 is the product of element #13 and element #2. It also indicates that this element has type 22 (2 a's and 2 b's), and shows the element as $(((ab)a)b)$. Obviously, the element's type is inherent in the last columns, however this column should make it easier to take a large table and pick out all elements of a certain type.

```
Basis Table:
1. 0 0 10 a
2. 0 0 01 b
3. 2 2 02 (bb)
4. 2 1 11 (ba)
5. 1 2 11 (ab)
6. 1 1 20 (aa)
7. 2 5 12 (b(ab))
8. 3 1 12 ((bb)a)
9. 4 2 12 ((ba)b)
10. 5 2 12 ((ab)b)
11. 1 5 21 (a(ab))
12. 4 1 21 ((ba)a)
13. 5 1 21 ((ab)a)
14. 6 2 21 ((aa)b)
15. 1 10 22 (a((ab)b))
16. 2 14 22 (b((aa)b))
17. 4 5 22 ((ba)(ab))
18. 5 5 22 ((ab)(ab))
19. 7 1 22 ((b(ab))a)
20. 8 1 22 (((bb)a)a)
21. 9 1 22 (((ba)b)a)
22. 10 1 22 (((ab)b)a)
23. 11 2 22 ((a(ab))b)
24. 12 2 22 (((ba)a)b)
25. 13 2 22 (((ab)a)b)
26. 14 2 22 (((aa)b)b)
```

A portion of the multiplication table for the same algebra is shown below. The table lists only the nonzero products of two basis elements. Coefficients are given in terms of the current field.

Assuming the default field GF(251) were in use, the table below can be interpreted as

$$
\begin{aligned}
b_1 b_1 &= b_6 \\
b_1 b_2 &= b_5 \\
b_1 b_3 &= b_{10} \\
b_1 b_4 &= -b_{11} + b_{13} + b_{14} \\
b_1 b_5 &= b_{11} \\
b_1 b_7 &= -b_{15} + b_{18} + b_{23} \\
b_1 b_8 &= -b_{15} + b_{22} + b_{26} \\
b_1 b_9 &= b_{25}
\end{aligned}
$$

Had we shown the entire multiplication table, we would have seen that $b_{13}b_2 = b_{25}$. Hence b_25 factors as $b_1 b_9$ and $b_{13}b_2$. This merely says that $a((ba)b) = ((ab)a)b$ in a right alternative ring. Recall that when Albert constructs an algebra in, say, $2a$'s and $2b$'s, products involving more than $2a$'s or more than $2b$'s will be zero. Other kinds of products, too, can be zero.

```
Multiplication table:
(b1)*(b1)
    1 b6
(b1)*(b2)
    1 b5
(b1)*(b3)
    1 b10
(b1)*(b4)
    250 b11 + 1 b13 + 1 b14
(b1)*(b5)
    1 b11
(b1)*(b7)
    250 b15 + 1 b18 + 1 b23
(b1)*(b7)
    250 b15 + 1 b22 + 1 b26
(b1)*(b9)
    1 b25
```

4. The configuration file

The configuration file contains definitions for making it easier to enter identities. This file can be edited by the user outside of Albert. Arbitrarily many definitions can be given. Long definitions can be continued on another line by placing a backslash (\) at the end of the line. The file can include blank lines. Comments begin with % and extend to the end of a line. A typical configuration file might look like this:

```
rightalt    (x,y,y)              % Right alternative law.
jordan      ((xx)y)x - (xx)(yx)
com         [x,y]
doublecom   [[x,y],z]

% A strange new identity:
ident3      (x,[x,y],x)
```

When Albert is initialized, the configuration file, is specified, is read into memory. Definitions occurring within this file can be used in subsequent commands, by surrounding the defined entity

with $'s. For example, one now could enter the command:

```
identity $jordan$ - $ident3$
```

The identity is interpreted as $((xx)y)x-(xx)(yx)-(x,[x,y],x)$. The configuration file can make use of definitions occurring elsewhere in the file. Moreover, a definition need not occur before its use. For example, one might have

```
RightAlt       (x,y,y)          % Right alternative law.
RightAltCom    [w, $RightAlt$]  % Right alternators commute.
```

This last identity is interpreted as $[w,(x,y,y)]$. Circular definitions will cause great problems.

5. Invoking Albert

Albert is invoked on the command line by giving its name followed by one optional argument:

```
albert [-f  filename]
albert -h
```

Here *filename* refers to the (relative of absolute) location of a configuration file. For example:

```
albert -f .albert
```

If no options are specified, no configuration file will be used. If invoked with `-h` option, Albert prints a short help message and quits.

6. Sample Scenario

The following example illustrates interaction with Albert. Text appearing after the `-->` prompt has been printed by the user; all other text has been typed by the Albert system.

```
[pasha@max-und-moritz] tmp--> albert -f /usr/local/albert/4.0M6/.albert
Albert version 4.0M6, 2019

Using /usr/local/albert/4.0M6/.albert

-->identity (x,x,y)

     (xx)y - x(xy)
Entered as identity 1.

-->identity (x,y,y)
    (xy)y - x(yy)
Entered as identity 2.

-->generators 2a2blc

Problem type stored.

-->display

Defining identities are:
1. (x,x,y)
2. (x,y,y)
Ground field is GF(251)
Problem type = [2a,2b,c]; Total degree = 5.
Multiplication table not present.
```

```
-->build

Building the Multiplication Table.

Build begun at Tue Jan 15 21:55:20 2019

Degree    Current Dimension   Elapsed Time(in seconds)
   1                 3                 0
   2                11                 0
   3                30                 0
   4                64                 0
   5                99                 0
Build completed.
Last Matrix 20.6% dense.

-->v m gap

T := EmptySCTable (99, Zero(GF(251)));
SetEntrySCTable (T, 1, 1, [1, 11]);
SetEntrySCTable (T, 1, 2, [1, 10]);
SetEntrySCTable (T, 1, 3, [1, 8]);
SetEntrySCTable (T, 1, 4, [250, 15, 1, 18, 1, 19]);
```

... (long output suppressed) ...

```
SetEntrySCTable (T, 64, 3, [1, 99]);
A := AlgebraByStructureConstants (GF(251), T);

-->polynomial (a,b,c)*[a,b]

Polynomial is an identity.

-->polynomial (a,b,(a,b,c)) + [b,a](a,b,c)

Polynomial is an identity.

-->polynomial [a,[b,(a,b,c)]]

Polynomial is not an identity.

-->remove 1

Identity 1 removed.
Destroyed the Multiplication Table.

-->generators 4a2b

Problem type changed.

-->display
Defining identities are:
1. (x,y,y)
```

```
Ground field is GF(251)
Problem type = [4a,2b]; Total degree = 6.
Multiplication table not present.
-->build

Building the Multiplication Table.

Build begun at Tue Jan 15 21:58:25 2019

Degree    Current Dimension   Elapsed Time(in seconds)
   1                 2                 0
   2                 6                 0
   3                15                 0
   4                35                 0
   5                77                 0
   6               146                 0
Build completed.
Last Matrix 7.8% dense.

-->poly (a,a,b)^2

Polynomial is not an identity.

-->poly (a,a,b)^3

Polynomial type not a subtype of Target_type.

-->quit
```

For further examples, and for illustration of a methodology for using Albert, see [HJ2], [HJK], [HJM], [J], [HJPS], [HP1], [HP2], [DZ], and [Z].

7. Polynomial Expression Language

The **identity**, **polynomial**, and **xpand** commands require a nonassociative polynomial to be entered. This section describes the proper syntax of nonassociative polynomials. In Appendix A, a formal grammar is given. Nonassociative polynomials are described using the following operators.

- addition: $x + y$.
- subtraction: $x - y$.
- unary minus: $-x$.
- scalar product: $3x$. The scalar is interpreted to be from the ring of integers.
- juxtaposed product: xy.
- commutator: $[x, y] = xy - yx$.
- associator: $(x, y, z) = (xy)z - x(yz)$.
- Jordan product: $x * y = xy + yx$.
- Jordan associator: $< x, y, z >= (x * y) * z - x * (y * z)$.
- Jacobi: $J(x, y, z) = (xy)z + (yz)x + (zx)y$.

- left associated exponential: $x^3 = (xx)x$. Warning: xy^3 means $((xy)(xy))(xy)$ not $x((yy)y)$.
- left/right multiplication: the general from of these expressions is $\{Ay_1y_2\ldots y_k\}$, where each y_i is of the form x‘ or x'. Here A and x can be more complicated expressions. x‘ denotes left multiplication by x, and x' denotes right multiplication. A sequence of such operators are applied from left to right, and surrounded by braces. For example, $\{xy‘z'u'\}$ denotes $((yx)z)u$, and $\{xy‘z'u'(w(ts))‘\}$ denotes $(w(ts))((yx)z)u$, and $\{xy'z'\}-\{zy‘x‘\}$ denotes $(xy)z-x(yz)$.
- artificial word: $W\{n;a:b:a:c:d\}$. Here n is called an *association type*. Often it is cumbersome to type in long parenthesized expressions. To simplify the typing, the artificial word construct can be used. $W\{n;a:b:a:c:d\}$ represents the word having letters a,b,a,c,d and association type n. Albert places a well-ordering on associations. If two associations w_1 and w_2 have different degrees, then $w_1 < w_2$ if w_1 has smaller degree. Suppose w_1 and w_2 have the same degree. If this degree is 1 or 2, then $w_1 = w_2$. But if

  $$w_1 = (l_1)(r_1)$$
  $$w_2 = (l_2)(r_2)$$

  then $w_1 < w_2$ if either $r_1 < r_2$, or $r_1 = r_2$ and $l_l < l_2$. Thus, the smallest degree n association type is

  $$(\ldots(((xx)x)x)x)\ldots)x,$$

  and the largest degree in association type is

  $$x(\ldots(x(x(x(xx))))\ldots).$$

  The degree 4 association types, in order, are:

  $$((xx)x)x$$
  $$(x(xx))x$$
  $$(xx)(xx)$$
  $$x((xx)x)$$
  $$x(x(xx))$$

  Thus $W\{4;a:a:c:b\}$ means $a((ac)b)$. Note that typing in a number n that exceeds the number of degree n association types is an error. The user can easily determine the number for an association using the **type** command described in the section *Command Language*.

The operators can be intermixed in any arbitrary fashion. For example, the following constructs are allowed:

$$(x,[x,y],x)$$
$$J(x,<x,y,z>,z)$$
$$[x^3,(x,x*y,y)]$$

All variables must be entered lower case letters. The order in which operators are applied, is controlled using parenthesis. Thus one may write $(x*y)*z$ or $x*(y*z)$. An ambiguous expression such as $x*y*z$ is illegal. Most expressions have the same common meaning as they do in mathematics. For example, scalar multiplication and unary minus $(-)$ have higher precedence over addition and subtraction, and therefore $3(a,b,c)+(c,b,a)$ means $(3(a,b,c))+(c,b,a)$. However, there are some caveats. When used in the presence of ^ or $*$, juxtaposition has higher precedence: xy^3 means $((xy)(xy))(xy)$ not $x(y^2y)$, and $xy*x$ means $(xy)*x$.

Appendix A: Formal Grammar

polynomial → *term*
| + *term*
| − *term*
| *polynomial* + term
| *polynomial* − term

term → *product*
| `int` *product*

product → *atom_or_double_atom*
| *atom_or_double_atom* ↑ `int`
| *atom_or_double_atom* ∗ *atom_or_double_atom*

atom_or_double_atom → *atom*
| *double_atom*

double_atom → *atom atom*

atom → `small_letter`
| *commutator*
| *associator*
| *jacobi*
| *jordan_associator*
| *artificial_word*
| *operator_product*
| (*polynomial*)

commutator → [*polynomial*, *polynomial*]

associator → (*polynomial*, *polynomial*, *polynomial*)

jacobi → J (*polynomial*, *polynomial*, *polynomial*)

jordan_associator → < *polynomial*, *polynomial*, *polynomial* >

artifcial_word → W { `int`; *letter_list* }

letter_list → `small_letter`
| *letter_list* : `small_letter`

operator_product → { *atom operator_list* }

operator_list → *operator*
| *operator_list operator*

operator → *atom*‘
| *atom*’

`int`: Positive. Sequence of digits not beginning with 0.

REFERENCES

[DZ] A. Dzhumadil'daev, P. Zusmanovich, *The alternative operad is not Koszul*, Experiment. Math. **20** (2011), no. 2, 138–144; Corrigendum: **21** (2012), no. 4, 418.

[JMO] D.P. Jacobs, S.V. Muddana, A.J. Offutt, *A computer algebra system for nonassociative identities*, Hadronic Mechanics and Nonpotential Interactions, Part 1 (ed. H.C. Myung), Nova Sci. Publ., 1993, 185–195.

[HJ1] I.R. Hentzel, D.P. Jacobs, *A dynamic programming method for building free algebras*, Comput. Math. Appl. **22** (1991), no.12, 61–66.

[HJ2] I.R. Hentzel, D.P. Jacobs, *A condition guaranteeing commutativity*, Intern. J. Algebra Comput. **2** (1992), no.3, 291–295.

[HJK] I.R. Hentzel, D.P. Jacobs, E. Kleinfeld, *Rings with* $(a,b,c) = (a,c,b)$ *and* $(a,[b,c],d) = 0$*: a case study using Albert*, Intern. J. Comp. Math. **49** (1993), no.1-2, 19–27.

[HJM] I.R. Hentzel, D.P. Jacobs, S.V. Muddana, *Experimenting with the identity* $(xy)z = y(zx)$, J. Symb. Comput. **16** (1993), no.3, 289-293; Erratum: **17** (1994), no.2, 213.

[HJPS] I.R. Hentzel, D.P. Jacobs, L.A. Peresi, S.R. Sverchkov, *Solvability of the ideal of all weight zero elements in Bernstein algebras*, Comm. Algebra **22** (1994), no.9, 3265–3275.

[HP1] I.R. Hentzel, L.A. Peresi, *The nucleus of the free alternative algebra*, Experiment. Math. **15** (2006), no.4, 445–470.

[HP2] I.R. Hentzel, L.A. Peresi, *Nuclear elements of degree* 6 *in the free alternative algebra*, Experiment. Math. **17** (2008), no.2, 245–255.

[J] D.P. Jacobs, *The Albert nonassociative algebra system: a progress report*, ISSAC '94 Proceedings of the International Symposium on Symbolic and Algebraic Computation, ACM, 1994, 41–44.

[Z] P. Zusmanovich, *Special and exceptional mock-Lie algebras*, Lin. Algebra Appl. **518** (2017), 79–96.